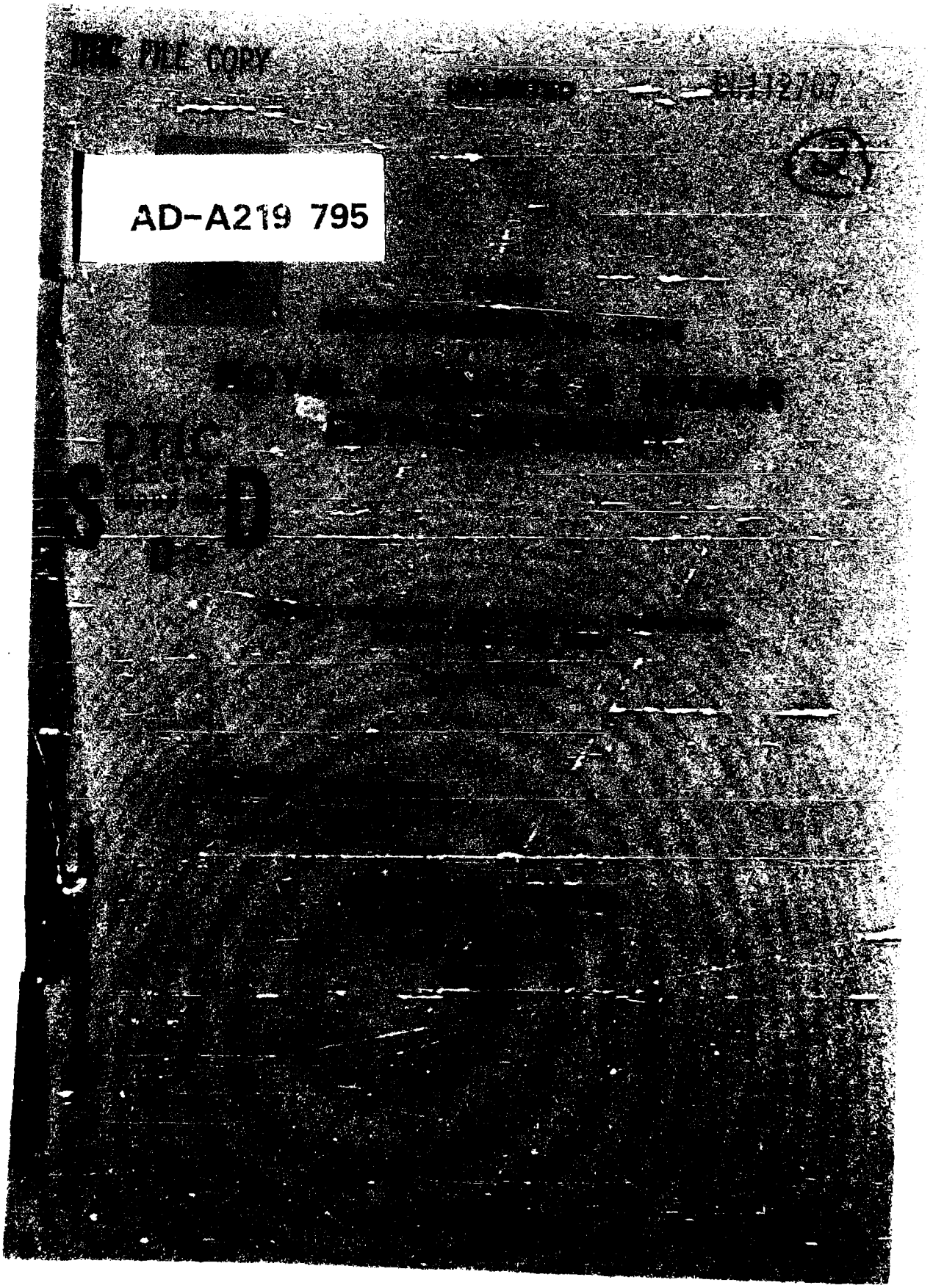

AD-A219 795

CONDITIONS OF RELEASE

0061790 BR-112707

********** U

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Royal Signals and Radar Establishment

Memorandum 4334

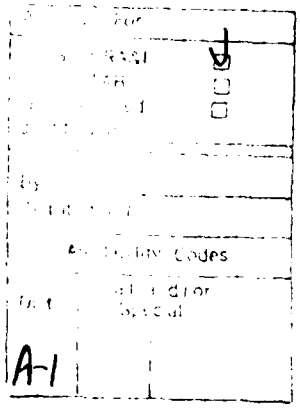
TITLE: Object Orientated Simulation on Transputer Arrays using Time Warp

AUTHOR: P. Simpson

DATE: December 1989

SUMMARY

The successful application of Transputers to distributed event driven heterogeneous simulation using the Time Warp methodology is demonstrated with transputers and occam providing a natural vehicle for this class of simulation. The simulation technique basically comprises a number of communicating simulation object processes, with appropriate action being taken to ensure the correct chronological sequence of processed simulation events. Time Warp is particularly attractive, since it permits all parts of a distributed processor network to operate in parallel (although some of the computation may later be undone). The need for hardware control of memory management has not been identified, although the requirement for a deadlock free, random point to point communications strategy has.



INTENTIONALLY BLANK

Contents

0 Abbreviations 6

1 Introduction 6

2 Distributed Event Driven Simulation 7

3 Time Warp Simulation 8

3.1 Rollback Mechanism 8

3.2 Simulation Output 9

3.3 Message Composition 10

3.4 Reproducibility 11

4 Transputer Array Implementation of Virtual Time Simulation 11

4.1 Controller Process (CP) Description 12

4.1.1 Messages to the HP 14

4.1.2 Output Options 14

4.1.3 Graphics Output 15

4.2 Detailed Host Process (HP) Description 15

4.2 Detailed Time Warp Code Description 16

4.3.1 Sequential Time Warp Process 16

4.3.2 Data Router 16

4.3.3 Time Warp Process 16

4.3.4 Output event 18

4.3.5 Process Event 21

4.3.6 GVT Estimation 22

4.3.7 Fossil Collection 23

4.3.8 Important Note 23

5 Low Level Operations 24

5.1 Global Virtual Time Estimation 24
5.2 Fossil Collection 25
5.3 Queue Data Structures 25

6 Message Passing Techniques **30**

6.1 Message routing 30
6.1.1 Routing Table setup 30
6.1.2 Message Broadcasting 33
6.2 Low level message passing procedures 34
6.2.1 Procedure route 34
6.2.2 Procedure array.in.four.out 35
6.2.3 Procedure four.in.array.out 35
6.2.4 Procedure analyse.network 36
6.3 Queue Handling Procedures 37
6.3.1 Procedure initialise.q 37
6.3.2 Procedure find.pos.to.insert.in.queue 38
6.3.3 Procedure insert.in.q 39
6.3.4 Procedure try.to.annihilate 40
6.3.5 Procedure select.next.q.event 43
6.4 Message Protocols 44
6.4.1 Protocol ps.time.warp 44
6.5 Libraries 48

7 Error Conditions **49**

8 Implementation Constraints **50**

9 Race Track Traffic Flow Example **51**

9.1 Input Data 52
9.2 Configuration 52

9.3 Memory Requirements 53

10 The Way Ahead **54**

11 Conclusion **54**

12 References **56**

13 Appendix **57**

0 Abbreviations

CP	Control Process
GP	Graphics Process
GVT	Global Virtual Time
HP	Host Process
ID	Identification Number
IQ	Input Queue
LVT	Local Virtual Time
OQ	Output Queue
PID	Process Identification Number
SCP	Switch Controller Process
SP	Simulation Process
SPA	Simulation Process Array
TDS	Transputer Development System
TW	Time Warp

1 Introduction

Simulations of large complex systems consume vast amounts of computing resources, often resulting in simulations running unacceptably slower than real time. Therefore the application of powerful parallel computer architectures to this problem area will be extremely welcome to the simulationist.

Currently there are two prime simulation methodologies, time driven and event driven. Time driven relies on the regular incremental update or time stepping of the simulation clock, and simulating the system up to this new clock time. It has advantages where for example the system has to be modelled essentially continuously, for example in the simulation of analogue circuits, and indeed greater accuracy may ensue with infinitely small time steps. However many simulation scenarios are not amenable to this methodology, in which for example the system may enter long periods of relative inactivity followed by much more intense activity periods. Here the time step would have to be tailored for active simulation periods therefore resulting in gross inefficiencies during relatively inactive periods.

The alternative approach to time driven is event driven simulation. Here the system clock is incremented at irregular steps, corresponding to the time of the nextsystem interaction activity (or event). Thus the clock advances through periods of simulation inactivity allowing processing resources to be concentrated in regions of interest in the simulation time domain. It is this methodology that we are particularly interested in, being appropriate to battlefield, traffic flow and communication network simulation scenarios.

It is widely accepted that parallel processing architectures offer significant increases

in computational performance over serial machines. Here we examine how a specific class of architecture based on reconfigurable arrays of transputers, the Supernode, can be applied to event driven simulation. We have used a prototype of the Supernode, known as the RSRE RAT (Reconfigurable Array of Transputers) cage for all algorithm development work, which comprises 16 worker T414 transputers, a transputer acting as a switch controller processor and a graphics processor. The system is configurable to any topology given the constraint of 16 transputers each with 4 communications links. The machine is hosted by an Inmos B004 processor as part of an IBM PC system.

2 Distributed Event Driven Simulation

Conventional event driven simulation in a serial processor is implemented by selecting chronologically events from a unique global event list, and activating and processing relevant objects according to the particular event. If necessary new generated events are reinserted in this event list for future scheduling and processing. This concept lends itself naturally to serial machines with global memory and a single thread of computer control. However in distributed processing environments, such as transputer arrays, where global memory is not provided, and we have multiple threads of process control this conventional approach is not possible.

To exploit parallel processing capabilities offered by a distributed architecture it is essential to action the processing of more than one event simultaneously. We therefore distribute simulation objects across the array and allow each object to advance its local logical time, driven by its own local event list. Objects requiring interaction with others must therefore communicate with other objects, implanting events into their respective local event list as appropriate.

Clearly, since the concept of global variables does not exist, the concept of a maintained global time is not permitted, in general, each simulation object will have its own local clock time, which in principle could be different to that of all other times in the system. It is important however, to ensure that events occur in chronological order to secure simulation correctness, such that a message corresponding to an interaction from an object A say, with a local time of t, arrives at object B, such that t is not in the simulation past of object B, (or if it does we must effectively undo any actions B may have initiated during the intervening period).

Currently two techniques exist providing effective time synchronisation across processor arrays. The first, known as null-message passing is a conservative process, relying on regular exchanges of time stamped messages between all interacting objects providing permission for receiving objects to advance their own local clocks to the new time, guaranteeing that past events will not later arrive. Whereas the second approach, known as Time Warping is based on optimistic processing, which makes no assumption about events arriving in chronological sequence. And here if an event does arrive in the receiving object's past, then the process back tracks in time, undoing any actions it may have

already done before processing the new event. Each object's local time is permitted to advance and retreat in time, hence the technique is often refered to as virtual time simulation.

3 Time Warp Simulation

Time Warp is a technique developed by Jefferson [1] for efficiently exploiting parallel distributed processing computer architectures with event driven simulation. The methodology has been adequately and elegantly described by Jefferson in the literature, so here only an overview will be provided.

Essentially simulation objects are distributed across the processor array and interactions between objects are triggered by passing relevant messages. Each object has associated with it various state variables, including simulation time - Local Virtual Time (LVT), which generally will be different for all simulation system objects. If an object decides to interact with another at a specific time, it initiates a message communication with the receiving object, sending information pertaining to the type of interaction, variables associated with the interaction, and most important, the anticipated time of influencing the receiver. Actually two time stamps are transmitted corresponding to this interaction time and the old interaction time in the sending object - which is not used. It is also permissible for an object to predict its own as well as other objects future events by inserting event messages in its own IQ. The receiving object will therefore accept the message with its interaction time and process it. This is fine if the requested interaction time is later than the receiving object's LVT, but this is not guaranteed and so if a message arrives with a time component earlier than the LVT of the receiving object, then the time of that object must be rewound to this earlier time, ensuring that any incorrectly initiated object interactions are undone. This mechanism is known as Rollback.

3.1 Rollback Mechanism

To activate Rollback and undo all possible incorrectly initiated actions it is necessary for all objects to maintain a history of all received messages, sent messages and local state variables. These are stored in data structures referred to as the input queue (IQ), output queue (OQ) and state queues respectively. The IQ and OQ therefore contain histories, arranged in chronological order of all received and sent messages respectively. The State Queue contains a record of all states associated with that particular object. In practice not all states will be required to be stored for memory economy reasons. The IQ is therefore analogous to the event list in a traditional event driven simulator, although here, distributed across the processor array with effectively each object having its own local event list.

Thus when a message arrives at an object with a time later than its LVT, it is simply inserted in the IQ to await future processing.

However if a message arrives with an earlier time than the current LVT, we must invoke Rollback which entails inserting the received event in the IQ as normal, setting the object's LVT to the new message time (hence 'Virtual' time) and resetting the object state to this new time by extracting the appropriate state variables from the state queue. This would be all that was required if the object had not already output messages to other objects in the meantime. We must therefore examine the OQ to establish what (if any) messages exist with timestamps later than the rollback trigger time and those that have not yet been sent, are deleted. But the effects caused by messages already sent must be undone using a concept of anti-messages. An anti-message is identical to the corresponding normal message, except that it has a flag set signifying it to be an antimessage. Whenever a positive and antimessage coexist at the same place they annihilate each other leaving no record of their existence. So to undo actions which may have been triggered by incorrectly sent positive messages from the OQ, we form and send their respective anti-messages to the appropriate objects which naturally leads to their destruction in the receiving objects IQ.

On receipt of an anti-message, if the positive message is unprocessed, i.e. it still exists in the IQ with a time later the the current object LVT, then it annihilates leaving no trace of the original message. If however the positive message has been processed, and therefore actioned possible further incorrect actions, the message still annihilates with the copy of the positive message in the IQ, but, since its time will be earlier than the current LVT, it triggers rollback just as receipt of a positive message earlier in time would.

Since a rollback can never be triggered earlier than the earliest LVT in the system this methodology guarantees that rollback are not infinite, and the overall system will advance in simulation time.

3.2 Simulation Output

In any simulation it is necessary to output sequences of events as and when they occur. Care has to be exercised with Time Warp to ensure that output events are valid and will not later be invalidated through a rollback. We therefore introduce the concept of Global Virtual Time (GVT). At any instant in the system there will be a number of objects, each with their own LVT, together with a number of time stamped messages in transit. Since the receipt of a message will not trigger an earlier rollback time, it is guaranteed that there is a minimum time threshold to which rollbacks can occur. This is GVT and is defined as the minimum of all object LVT's and time stamps of all messages in transit (i.e messages sent but not yet received). Events in the system up to GVT are all therefore valid and will never be rolled back upon, and can be used for output purposes. Additionally these events can then and only then be deleted from the system queue data structures if required, releasing memory, a process known as fossil collection.

The Time Warp technique has been implemented fully on a Supernode prototype and validated using the flow of vehicles on a simple road network as a demonstration example.

3.3 Message Composition

Messages in this kind of distributed simulation system correspond with events in a traditional event driven simulator, with a significant difference however. If for example, an event represents the interaction of two objects such as a tank and a missile, then the missile arrival event must also bear information such as missile speed and trajectory, so that all relevant tank state information can be appropriately updated. Alternatively, if an event is the collision between two vehicles, the event message must convey certain knowledge of the colliding car such as velocity. Therefore, in the context of Distributed Event Driven Simulation, messages are not just events, but also contain relevant, sending object information allowing the receiving object to update all its relevant state variables as necessary. Remembering that this information is not directly accessible in a distributed processing and distributed memory environment.

In the race track example discussed in detail later and used as a test bed for virtual time simulation, messages represent the passage of vehicles from one road segment object to another, and contain information such as vehicle velocity to determine future behaviour in the next road segment together with states associated with the vehicle such as fuel resources etc. This scenario could be represented differently by for example representing the vehicles as objects.

Consider a more complex interaction such as a tank firing a shell at another. This is representable in more than one way - a possible scheme is to represent both tanks as objects and the shell as message between them. Thus the message will contain information such as trajectory velocity and so on. On arrival of the shell with its associated property states at the receiving tank relevant internal states are updated - related to damage received perhaps. However an alternative modelling could use three separate objects i.e 2 tanks and a shell object. Now when tank 1 wants to initiate firing it sends a message to the shell object, which then sends a message to to the second tank object.

The choice of implementation depends largely on the availability and manageability of available process resources. For example if each shell in the system (or perhaps each bullet) were to be represented as separate objects, then for most of simulation time many of the objects will be inactive. Thus if objects can be dynamically created and destroyed and allocated to physical processors to balance processor load this approach may be optimum, as opposed to treating these transient object types as individual messages, with their own internal states.

Since memory management facilities, providing dynamic object creation and destruction are not available on transputers, we have adopted the approach of messages representing certain, more mobile or temporary object types, with messages containing all relevant state variables (- which may be updated by objects and information allowing receiving objects to update their appropriate state variables).

3.4 Reproducibility

It is important that event sequences produced by different simulation runs are identical given identical initial conditions. We must therefore ensure that if more than one event arrives with identical time components in a particular object, the logical time ordering of events is deterministic and reproducible. To provide this facility we have introduced a concept of micro time. This guarantees that event messages will have unique time stamps for event scheduling even though they may have similar logical times. Microtime is introduced by adding a unique fractional time component to all integer time stamps in the system, which is used to maintain a unique chronological order within the time warp input and output queue data structures.

4 Transputer Array Implementation of Virtual Time Simulation

The current implementation of the Virtual Time Simulator on an array of transputers, is as far as possible independent of processor array topology (which in our case is determined by the setting of a switch through which all transputer links are connected) and the process : processor ratio. This independence is achieved through the user definition of process connectivity tables and appropriate process placement at compile time.

The simulation system consists of 4 distinct process types; a host process (HP), a controller process (CP), a graphics process (GP) and a time warp simulation process (SP) arranged as shown in Figure 1. Any number of time warp processes can be used distributed over any number of available processors, the upper limit being controlled by available processor memory. In the race track implementation example given later, 38 time warp processes are configured over 15 processors.

The HP runs on the B004 transputer and handles all screen and file output together with data input during simulation initialisation. Connected to the HP via the Switch Controller Transputer (SCP), is the CP which is responsible for initialisation, controlling Global Virtual Time (GVT) estimation within the simulation array, and routing simulation output to the HP or Graphics Process (GP) running on an Inmos B007 system as required. The remaining 15 RSRE Rat Cage Transputers may be configured in any way and used to execute as many copies of the time warp process as the user application demands.

The following subsections contain full specifications of each process type mapped onto individual processors. Block diagrams for the entire system structure, indicating the heirarchy of processes etc are provided in the Appendix.

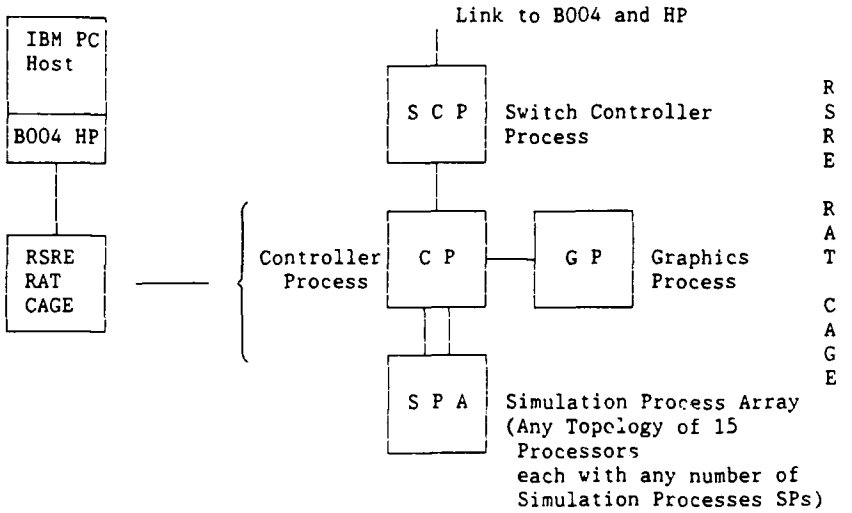


Figure 1: Block Diagram of Main Processing Components

4.1 Controller Process (CP) Description

The Controller Process (CP) resides on processor number 16 and is primarily responsible for acting as an interface between the SPA and the HP and GP, and for generating appropriate interrupt sequences for GVT estimation and fossil collection. It is contained within a separate compilation unit (*SC Controller*) and is the only process on this processor. Its 4 I/O channels are mapped directly on to the processor links, with links 0 and 3 providing communication with the HP (via the SCP) and GP respectively, and links 2 and 3 communication with the SPA.

Because in the current occam implementation it is more convenient to access distinct channels (rather than arrays of channels) externally to the SC (because of easier configuration) and arrays of channels internally, procedures *array.in.four.out* and *four.in.array.out* run in parallel within the SC to provide the necessary channel access conversions.

The CP first of all awaits a signal from the HP (*start.and.analise*), together with two run time defined constants defining the extent of output (*output.opt*) and a user defined real time delay - specified in milliseconds, between graphics display updates (*graphics.delay*). The CP acknowledges the HP with introductory messages and enters the message routing table setup procedure *analise.network*, which synchronises with the corresponding procedures running in each of the SP's. The routing vector for the CP is then sent too, and displayed by the HP. Various other initialisation actions then follow, firstly the

graphics display backdrop is read in to the CP and passed on to the GP and displayed, followed by the initialising event list, which in the 'race track' example contains details of all vehicles in the system. These are sent to the appropriate SP by a sub-process running in parallel with the main message handling and array control process.

An occam PRI ALT construction provides the main message handling and simulation array control functions. GVT estimation and fossil collection are controlled by the high priority input process to generate interrupts every *gvt.freq* milliseconds using occam timer facilities, and the low priority inputs deals with those messages received back from the SPA.

Currently the CP caters for the receipt of 10 message types (occam protocols) from the SPA:-

gvt.req.broad.p : After a user defined timeout period (*gvt.freq*), the CP generates an interrupt broadcast message requesting the array close down for GVT estimation and fossil collection. As explained in Section 6.1.2 the broadcast procedure sends messages down each connected channel and waits for the arrival of their 'echos' of receipt. This simple protocol handles the receipt of this broadcast message echo of the request to each SP to suspend output.

lvt.req.broad.p : This is the echo of the CP generated broadcast to each SP to send their current LVT

gvt.broad.p : As above, but the echo of the broadcast of GVT to each SP in the array.

ok.to.cont.broad.p : As above, but the echo of the broadcast giving permission for all SP's to resume normal processing after fossil collection. (This is not strictly necessary, although required here as the routing network is insufficiently robust)

gvt.pause.ack.p : Following successful event output suspension and acknowledgement of receipt of all sent messages, each SP responds with the signal 'gvt.pause.ack.p' to the CP. The CP therefore awaits for signals from ALL SP's and then broadcasts a request for each SP's LVT using the *lvt.req.broad.p* protocol.

lvt.ack.p : The LVTs from each SP return to the CP as part of the *lvt.ack.p* protocol. When the complete set of LVTs from all SP have been received, the CP computes GVT ($GVT = min\{LVT_i\}$) and broadcasts it to the SPA with the *gvt.broad.p* protocol.

fc.event.route.p : Fossil collection commences in each SP on receipt of GVT, during which time each SP empties its OQ of events earlier than GVT, routing them (via the *fc.event.route.p* protocol) to the CP for possible display and output. This input option stores sequences of events from each SP in the array structure *event.history*, and if the appropriate output option is set also routes the 'raw' events back to the HP.

fc.end.of.event.p : This signal is generated by each SP to denote termination of fossil collection in that particular SP. When the set of signals from all SPs has been received, either the event history is ordered and sent to the HP or(and) sent to the GP for display. Following either display or host output the signal *ok.to.cont.broad.p* is broadcast back to the SPA, granting permission for each SP to resume normal processing.

event.route.p : Currently events are routed throughout the SPA and CP using this protocol. This process therefore provides the necessary routing capabilities, routing events from channel 1 to 2 or vice verca.

ps.prot.tt.out.string : This protocol is used to route diagnostic messages from the SPA to the HP, although in the race track example is not used.

Within the CP all other message protocols are invalid, and receipt of which will result in process termination (via a call to procedure *crash*).

4.1.1 Messages to the HP

Output messages to the HP are generated with derivatives of Inmos i/o routines (*ps.write.full.string*, *ps.newline* and *ps.write.int*). These take identical parameters to their Inmos counterparts, but strictly adhere to the protocol *ps.prot.tt.out.string* defined in *ps.time.warp*, with which all main channels in the TW system are defined. The need for these procedures arises because the Inmos routines assume channels with trivial protocol definitions (CHAN OF BYTE) and are not compatible with more complex protocols.

4.1.2 Output Options

Part of the initialisation procedure is the user definition of an output option defining the extent of event output. In the current implementation this is a 32 bit integer with individual bits set according to particular desired options as defined below:-

Bit 1 : (LSB) If set all events are sent from the CP to the HP for display. The events are unordered, and sent to the HP in the order in which they are received by the CP during fossil collection.

Bit 2 : If set, events are sent to the HP in time increasing order for each object in the TW simulator.

Bit 3 : If set, events are displayed by the GP, in time increasing order. (See Graphics Output below)

All other bits have no effect

For example an output option of 5 will send unordered events on the HP and display time ordered events on the GP

4.1.3 Graphics Output

Graphics output only occurs if bit 3 of the output option control word is set. Events are selected from the event history array in increasing time order, and sent to the GP. The user can define a delay (in milliseconds) between GP updates by setting *graphics.delay* to a value greater than zero.

4.2 Detailed Host Process (HP) Description

The Host Process (HP) executes on the B004 transputer located within the host PC system. The occam code is part of an 'EXE' code fold and is primarily responsible for passing initialisation information to the SPA (via the SCP) and for intercepting, displaying and storing returned diagnostic information.

Its functionality is shown in the Appendix and basically comprises an initialisation phase, where file and keyboard data is read and a phase where all messages from the CP are intercepted and displayed on the host screen and stored in an output file if required.

The initialisation provides the user with the opportunity to direct all output to a diagnostics file (TDS fold) as well as to the Host screen, and requests input for the values of *output.opt* and *graphics.delay*. The output option is an integer with bits set according to whether ordered or unordered events are to be displayed by the HP or whether events are to be displayed by the GP (or any combination). And the user can if desired specify a real time delay (in milliseconds) between graphics display updates by setting a positive value to *graphics.delay*.

Finally 5 processes execute in parallel, three of which split all output to screen and file driving processes, whereas the other two send and receive messages to and from the network. All messages to the CP are all for initialisation, a *start.and.analise* signal to trigger the commencement of network analysis (via procedure *analise.network*), and data corresponding to the graphics backdrop and initial event list - both of which are read in from a file (TDS fold). The main data input process accepts data from the CP or from the Host keyboard via an occam ALT construction. SPA data is routed to the screen handling process and the filer process if required, whereas in the current implementation, input from the keyboard triggers process termination. In future implementations this interface could provide a mechanism for changing run time parameters, introducing new events or changing output options and so on.

4.3 Detailed Time Warp Code Description

The main TW occam code is encapsulated within an 'SC' compilation unit with the descriptor *ps.time.warp.object*, containing procedure *tw.object*. Multiple process to processor allocation is achieved by combining multiple calls of this separate compilation unit, and is exemplified in the 'race track' example with 2 and 3 process to processor allocations in procedures *two.proc.process* and *three.proc.process* respectively. Inter process channels have to be declared by the user within these units, and care should be taken to ensure agreement with the user set up *conn.table* connectivity table.

tw.object contains together with the main TW process, routines, executing in parallel providing more convenient channel access, and a procedure for automatically generating route tables (procedure *analise.network*). Network analysis and the main TW process run sequentially, although both run in parallel with the channel access procedures.

The two channel access routines have been provided because at code levels above this, it is more convenient to have access to district channels for configuration purposes, whereas at lower levels, access to arrays of I/O channels is more preferential.

4.3.1 Sequential Time Warp Process

This procedure is a combination of processes for data routing and the main TW process, both of which execute in parallel.

4.3.2 Data Router

Two parallel processes form the data router, an input router and an output router. Incoming messages are intercepted by the input router and either directed to the the Time Warp process, if this is its intended destination, or passed to the output router where decisions are made according to *route.table* to establish the appropriate output data channel. The input router also provides the functionality associated with data broadcasting.

The output router therefore takes input from either the output channel of the TW process, of from the input router and outputs to the appropriate output channel.

4.3.3 Time Warp Process

The main TW process executes in parallel with both the input and output router processes, and has a main input channel from the input router and an output channel to the output router. A block diagram of its functionality is given in the Appendix.

During each cycle through the main code section, any available input is processed.

else, if no input is available, other lower priority actions are taken. The structure is implemented via an occam PRI ALT construction, with data input the high priority process, and guarded clauses forming the remaining low priority processes.

Data Input : Data available on the input channel is always processed at high priority, with various actions being taken according to the input data type. currently there are 6 input data types, 4 of which are associated with GVT estimation, and the remaining two correspond to event message arrival, and acknowledgements to previously sent messages. It is important that this process should always be available for input to prevent process blocking.

Event Message Arrival : Event messages arrive in the TW process from the input router via the *event.p* protocol. On receipt, an immediate check is made to ensure the event is at its correct destination - if not the transputer error flag is set (with a call to procedure *crash*) and the system terminates. Assuming events have been correctly routed, an acknowledgement of receipt is generated and sent to the sender via the output router with the *event.rec.ack.p* protocol.

In this implementation the CP injects initialising in to the SPA and does not currently have the capability to receive event acknowledgements. Therefore to prevent their generation the CP sets the sender and receiver message data fields equal, and on receipt of event messages, the tw process checks for this equality, and if so suppresses acknowledgement generation. This condition is only ever true for initialising events, although in future implementations, this imposition may be a restriction - if we require to route message events to the IQ of the sending process for example. A simple fix would be to always generate acknowledgements and provide the CP with the capability of receiving acknowledgements.

If the received message is an antimessage, an attempt is made to annihilate it with its positive message in the IQ, assuming it exists, with a call to the procedure *try.to.annihilate*. In the current implementation, the routing network will always ensure that messages arrive in the order in which they are sent, therefore, when an antimessage does arrive it must always annihilate with its positive message. If the IQ empties after annihilation the flag *waiting.for.more.input* is set FALSE , to prevent repeated (and uncontrolled) selection attempts from an already empty queue (see later). If a positive message arrives, and therefore does not annihilate, the event is inserted in the IQ in increasing chronological order via a call to procedure *find.pos.to.insert.in.q*, and repeated calls to *insert.in.q* to store each message component in the appropriate field of the IQ data structure. Both receive and send time fields of the IQ structure are set to the receive time of the input message - which is actually the send time component of the message as it left the sending process. The current code will crash, if at this stage the IQ fills, although this problem can be avoided be either adopting larger queue structures, estimating GVT more frequently (hence activating fossil collection more often) or adopting the flow control techniques as discussed by Jefferson [1].

The process will also crash during this input phase, if either an antimessage or

a duplicate message is inserted in the IQ. These checks where incorporated for debugging purposes only, and can be removed if performance becomes an important issue.

Finally during the input phase, a test is made to establish if the receive time of the inserted message is earlier than the current LVT, if so rollback is triggered by setting *rollback.flag* TRUE, and resetting LVT to this new earlier time. In addition, several pointers associated with the OQ are reset - these will be discussed later, and any states associated with the TW object are restored to the position just prior to this new LVT. (In the race track example, the only state is the number of passing vehicles, given by by *n.pass*).

Event Receipt Acknowledgement Signal : Acknowledgements are generated immediately by the receiving TW object and routed to the sender on receipt of an event message via the *event.rec.ack.p* protocol. The acknowledgement receiver therefore simply decrements a counter - *n.en.route* corresponding to the number of sent events currently in transit.

GVT Estimation signal inputs : The remaining four input data protocols are concerned with GVT estimation. At periodic intervals (given by *gvt.freq*) the CP initiates a sequence of interactions to allow estimation of GVT. (see Section 5.1).

The first message received during GVT estimation requests that each TW object process suspend outputting events and await receipt of all sent, but unacknowledged messages. (via protocol *gvt.req.broad.p*). The input process immediately sets *gvt.pause.req* TRUE and *ack.gvt.pause* FALSE, the significance of which will be discussed later.

The second signal from the CP occurs when all SP's have flushed their output buffers and suspended output, and is the request for each object's current LVT. The input process responds by sending LVT to the CP via the output router process as part of the *lvt.ack.p protocol.*

Following receipt of all LVT's, the CP computes GVT, broadcasts this back to each SP with the *gvt.broad.p*, where the flag *fc.going* is set TRUE, to trigger the start of fossil collection (see below).

Finally when fossil collection has finished, the CP sends a signal to all SP's (*ok.to.cont.broad.p*) to allow the resumption of normal processing by setting *gvt.pause.req* to FALSE.

If no input data is available, then depending on the states of various flags, one of four actions is occurs (if possible).

4.3.4 Output event

Events are output only if the ALT guard *ok.to.output* is TRUE, which is reset at each cycle through the code. It is incorrect to attempt to output events if the OQ is empty, although if rollback is underway the code section should be entered to ensure that

pointers are correctly reset, or if the number of events in transit (*n.en.route*) exceeds a user defined threshold (given by *max.no.not.acked* defined in library *twlib.tsr* - logical name *ps.twlib*), or if GVT estimation has been triggered. *ok.to.output* is therefore a function of various logical conditions, as given below.

Let	A = ok.to.output	
	B = oq.empty	TRUE if OQ is empty
	C = rollback.flag	TRUE if rollback is underway
	D = n.en.route < max.no.not.acked	TRUE if number of messages in transit is less than max.no.not.acked
	E = gvt.pause.req	TRUE if GVT estimation triggered

Then

$$A = (\neg B \vee C) \wedge D \wedge E$$

If therefore the flag *ok.to.output* is TRUE and if no input action is possible, then an output attempt is made, with different actions taken according to whether rollback has been triggered or not.

The OQ contains events that have been previously selected from the IQ, processed and inserted in the OQ (see later), and has associated with it pointers corresponding to the last sent event. Specifically *output.q.next.event.pointer* contains a reference to *output.q.pointer*, the position of the last sent message whose time is given by *output.q.sel.time*.

If the rollback flag is not set, then an attempt is made to select the next chronological event from the OQ with a call to procedure *select.next.q.event*. If an event is selected (i.e. the previously selected event wasnot the last in the OQ), it is output to the output router using the *event.p* protocol. Pointers *output.q.sel.time* and *output.q.next.event.pointer* are updated by the selection procedure. A flag associated with the OQ is set TRUE to indicate the selected message has been sent, and the counter of the number of events in transit is incremented. The event is not deleted from the OQ since it may be required by a subsequent rollback. If an event is not selected the flag *oq.empty* is set TRUE, which is a function of the flag *ok.to.proces'*, to prevent repeated unsuccessful selection attempts.

However, if rollback has been activated, a different course of action is required. In the input process, rollback is triggered when an event arrives with a time earlier than the current LVT. This process also sets *rb.nxt.ptr* to -1 and *rb.sel.time* to the time of the message which triggered rollback, pointers which are now used to select rollback events (antimessages) from the OQ. Thus starting at the rollback time, events are successively selected, and deleted forward in time to the final event in the OQ. If the selected event has been sent (as indicated by the flag stored in the *output.q.sent.flag* array), its antimessage is produced and output (*n.en.route* is incremented), else if the positive message has not

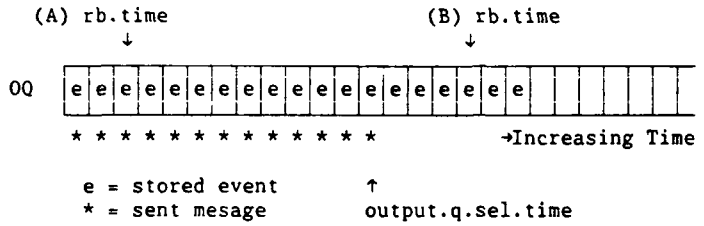


Figure 2:

yet been sent it is deleted only (using procedure *try.to.annihilate*). If for any reason the annihilation procedure fails to delete the positive message selected from the OQ, the the program will terminate via a call to procedure *crash*.

To resume outputting positive event messages at the correct position after rollback it is necessary to store two reset pointers. These are *rb.reset.time* and *rb.reset.ptr*. Their values depend on the location of the last output event in the OQ during non-rollback, as given by the value of *output.q.sel.time*, with respect to the value of the rollback trigger time (*rb.time*). This is exemplified in Figure 2.

Consider rollback occurring at A ; it is necessary to delete all later OQ events and send antimessages of events already sent (indicated by *) and then reset *output.q.sel.time* to the triggering rollback time. But if rollback B occurs, later events are deleted only, with *output.q.sel.time* remaining as it is. Thus during event input, if rollback is triggered, then the rollback time must be compared with the current value of *output.q.sel.time*, to set *rb.reset.time* (and *rb.reset.ptr*) as appropriate.

The instruction flow through the code is such that after each attempt to output events (i.e. *ok.to.output* is TRUE), control is always returned to test for any available inputs. It is therefore possible for messages with earlier times to arrive and trigger a new rollback during one already proceeding. This condition is dealt with by essentially permitting rollback triggering events to cancel with each other, with the net result to use only the earliest rollback time. As each new rollback is triggered (i.e. messages arrive with a time less than the current LVT) we update *rb.sel.time* to this new value and set *rb.next.ptr* to -1. As described above, we also check the relationship between the position of this new rollback time with the current non-rollback event selection time (*output.q.sel.time*) and update *rb.reset.time* and *rb.reset.ptr* as appropriate.

As events are deleted from the OQ during rollback, it is necessary to examine the position of the new vacancy with respect to *rb.reset.time* - the time at which normal selections occur after rollback, if it occurs before rb.reset.ptr then the pointer must be decremented. The annihilation procedure automatically decrements the normal selection pointer (*output.q.sel.time*) if necessary, but it is the user's responsibility to correctly reset

other queue associated pointers.

Rollback terminates when the OQ selection procedure fails to select an event from the OQ, for deletion and/or sending as an anti-message, because the selection has gone off the queue end, at which point *rollback.flag* is set FALSE and the two pointers, *output.q.sel.time* and *output.q.next.event.pointer*, used to control normal event selection reset to *rb.reset.time* and *rb.reset.ptr* respectively.

4.3.5 Process Event

Events are selected and processed from the IQ, only if there is no available input data, or output is currently inhibited (*ok.to.output* equal FALSE), and only if other conditions used to define the flag *ok.to.process* are satisfied. Specifically attempts are made to select events from the IQ for processing, only if IQ contains unprocessed events (i.e. *waiting.for.more.input* is FALSE), rollback is not in progress - it is undesirable to insert events in the OQ during rollback - unless very special care is taken various reset pointers, if fossil collection is not underway (see later) or if a request has not been received to shut down for GVT estimation. In future implementations, it may be possible to relax some of these conditions which have been imposed primarily for safety reasons.

Hence during each cycle through the code, the flag *ok.to.process* is set being the logical 'AND' of the inverse of *waiting.for.more.input*, *rollback.flag*, *gvt.pause.rec* and *fc.going*.

i.e. Let A = ok.to.output
B = waiting.for.more.input
C = rollback.flag
D = gvt.pause.req
E = fc.going

Then

$$A = \neg B \wedge \neg C \wedge \neg D \wedge \neg E$$

If *ok.to.process* is TRUE, the next unprocessed event is selected from the IQ via a call to *select.next.q.event* with selection pointers LVT and *input.q.next.event.pointer*, processed and inserted in the OQ. If the selection process goes past its last queue event the flag *waiting.for.more.input* is set TRUE to prevent repeated abortive attempts at unsuccessful selection. It is set FALSE on insertion of a new event in the IQ. Also, if during event insertion in the OQ a duplicate event is detected, or the queue fills, program termination will occur.

The processing phase represents the interaction of the event message with the object represented by the process - in the race track example objects are road segments and

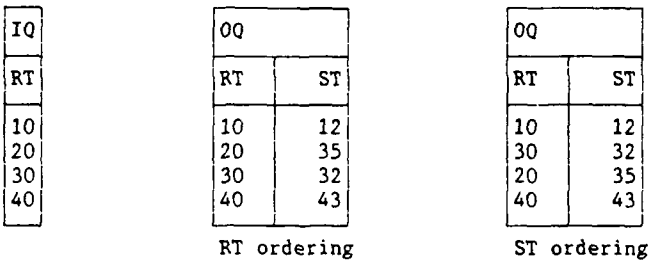


Figure 3:

event messages represent vehicles. Thus in this example, the interaction processing is trivial, just requiring the update of vehicle position, its send time i.e. the time at which it leaves the road segment object and the next object destination field. The number of vehicles which pass through the road segment - *n.pass* is incremented, and stored the the state data queue structure, this is the only state variable associated with the object in this rather trivial example. In a more realistic simulation object, this processing phase will dominate overall processing and there would probably be several tens of object state variable to store.

Events are selected chronologically from the IQ and the object's LVT is updated immediately to the receive time of this newly selected message. The process computes the new send time (referred to in the code as *output.lvt*), and the modified event inserted in the OQ using the event receive time (RT) rather than its send time (ST), to maintain chronological order. This is vital to allow easier event selection from the OQ during rollback, as shown in the Figure 3. For example, if rollback is triggered with a time of 25, with RT ordering it is easier to select the correct events with RT's of 30 and 40 than in the case with ST ordering.

4.3.6 GVT Estimation

On receipt of the CP generated signal in each SP to stop outputting event messages, (*gvt.req.broad.p*), the flag *gvt.pause.req* is set TRUE. This has the immediate effect of suspending output and further processing (*ok.to.output* and *ok.to.process* are set FALSE). At the start of each cycle through this part of the code, the flag *ok.to.ack.pause* if defined, and only becomes TRUE when the number of un-acknowledged messages (*n.en.route*) is zero. This immediately triggers an acknowledgement back to the CP denoting close

down of the SP (via *gvt.pause.ack.p*), and the flag *ack.gvt.pause* is set TRUE to prevent repeated generation of acknowledgements.

On receipt of the request for LVT from the CP (*lvt.req.broad.p*), and *fc.going* is set to signify the commencement of fossil collection.

4.3.7 Fossil Collection

During fossil collection, events are successively selected from the OQ from the earliest event to the event in the OQ prior to GVT, output to the CP and annihilate. As before use is made of procedures *select.next.q.event* and *try.to.annihilate*. Selection is made with pointers *fc.sel.time* and *fc.event.pointer*, which are both initialised to -1 to allow selection from the earliest event. Since fossil collection can occur during the course of rollback, it is necessary to update pointers *rb.reset.pointer* and *rb.nxt.ptr*, if events are annihilated from positions earlier in the OQ. In practice this will always be the case since rollback times are guaranteed always to be greater than GVT, and hence FC selection time.

The corresponding events are also deleted from the IQ, and if for any reason selected events are not annihilated processing will terminate.

Note.

The current code assumes that there is always a one-to one correspondence between events in the IQ and OQ, if this condition is not met, a more general IQ event selection and deletion procedure must be developed

When all events up to (but not including) GVT have been output and removed from the IQ and OQ, the fossil collection process sends a signal (*fc.end.of.event.p*) to the CP denoting fossil collection termination, and the flag *fc.going* is set FALSE to prevent further fossil collection selection attempts.

The CP, on receipt of all *fc.end.of.event.p* signals from all SP's responds with *ok.to.broad.p* which when received by the input process, resets *gvt.pause.req* to FALSE, immediately allowing the recommencement of event output and processing - if possible.

In future implementations it should be possible to resume normal processing (by setting *gvt.pause.req* to FALSE) immediately on fossil collection termination, although practice indicates the requirement for a more robust, deadlock free routing network.

4.3.8 Important Note

Extreme care should be exercised deleting events from the OQ, as the pointers *rb.reset.ptr*, *rb.reset.time*, *rb.nxt.ptr* and *rb.sel.time* may also require updating. These are not updated by the procedure *try.to.annihilate*. This procedure updates only those pointer associated directly with the queue structures.

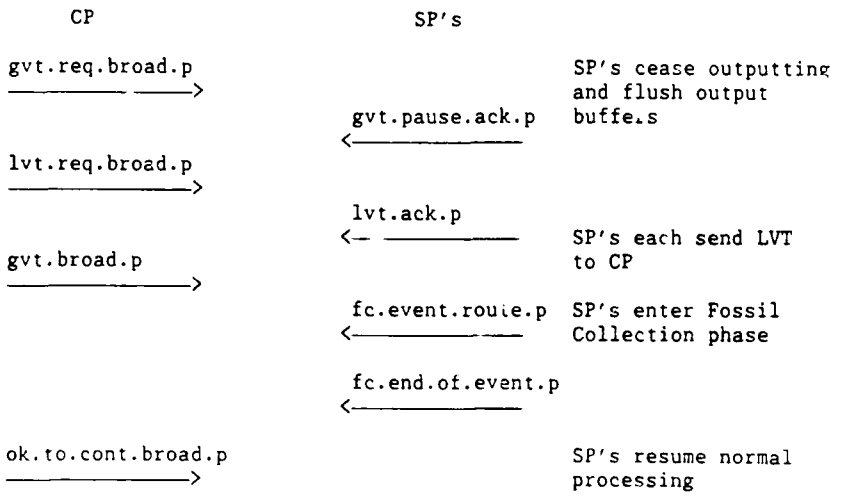


Figure 4: GVT Estimation and Fossil Collection Protocol

5 Low Level Operations

5.1 Global Virtual Time Estimation

In order to accurately compute GVT, which is defined as the minimum of all simulation process LVT's (assuming no unprocessed messages are in transit) it is necessary to bring the SPA to a controlled halt. The current implementation achieves this via a complex message exchange between the CP and each of the SP's. In future implementations, it may be more efficient to use the Supernode's control bus for example.

The GVT message exchange is given in Figure 4, and is initiated by the CP after a user defined timeout, given by the constant *gvt.freq*. (This is defined in library *gvt.tsr* - logical name *ps.gvt*). At this instant the CP broadcasts a signal using the *gvt.req.broad.p* protocol to all SP's. On receipt, each TW simulation process suspends outputting event messages (by setting *gvt.pause.req* to TRUE) although continues to receive and acknowledge any input messages, and acknowledgements to previously sent messages. When all sent messages have been acknowledged (i.e. when *flushed* becomes set TRUE) a signal is returned to the controller, signifying that that particular process has successfully suspended all output, and that all sent messages have been received at their respective destinations (via the protocol *gvt.pause.ack.p*). Each process therefore suspends activity in a controlled manner, guaranteeing that no messages become 'stuck' in transit.

The CP meanwhile awaits the set of *gvt.pause.ack.p*'s from the SPA and responds with a request to each SP to send its respective LVT, via the *lvt.req.broad.p* protocol. Each

SP responds to receipt of this broadcast signal with its own LVT (using the *lvt.ack.p* protocol). And in a similar manner as before, the CP awaits the set of LVT's, from which it computes GVT (defined a the minimum of the set of received LVT's) and broadcasts GVT back to the SPA via the *gvt.broad.p* protocol.

GVT is the limit to which it is guaranteed that no further rollback in the system will go beyond, thus any event stored in the system with a time earlier than GVT will be valid and not later annihilated through a rollback mechanism. The broadcast of GVT therefore acts a signal to commence fossil collection (*fc.going* is set TRUE), the procedure in which event queues are purged to free memory and valid events are output to the snapshot collector running in the CP. Normal processing is then resumed, when all SP's have completed fossil collection. (*gvt.pause.req* is set FALSE).

5.2 Fossil Collection

Fossil collection is triggered on receipt of the latest GVT from the CP, and acts as a mechanism for displaying and outputting valid events in the simulation system and also provides a mechanism for releasing memory no longer required.

When fossil collection is triggered, events are selected chronologically from the earliest event to the event occurring prior to GVT from each of the OQ's and output using the *fc.event.route.p* protocol. The signal *fc.end.of.event.p* is then sent to the CP to denote that all events up to GVT have been sent by that particular SP. The CP meanwhile awaits the set of all events from all SP's to commence snapshot generation. During fossil collection, as events are selected and removed from each OQ, the corresponding event is deleted also from the IQ. In this implementation a run time error will result if the corresponding event is not located in the IQ. It is therefore important that there should always be a one-to-one correspondence between events in the IQ and OQ. In future implementations, if this constraint cannot be met for what ever reason, then a more complex selection and deletion strategy will be required.

Following the receipt of the set of *fc.end.of.event.p* from each SP denoting that all events up to GVT have been sent to the CP and deleted from the state queues, the CP broadcasts *ok.to.cont.broad.p*, which gives permission for each SP to resume normal processing. To improve performance, it should be possible to resume normal TW processing immediately it finishes fossil collection, rather than waiting for all processes to finish. This has not been implemented as it requires a more robust, and totally non blocking message routing system.

5.3 Queue Data Structures

Each queue is defined by a number of data arrays and scalars:-

[x.queue.size'INT x.q.rec.time

	[x.queue.size]	INT x.q.send.time
	[x.queue.size]	INT x.q.sender
	[x.queue.size]	INT x.q.recever
	[x.queue.size]	INT x.q.sign
[x.queue.n.states]	[x.queue.size]	INT x.q.state
	[x.queue.size]	INT x.q.pointer
	[x.queue.size]	INT x.q.free.list
		INT x.q.last
		INT x.q.next.free
		INT x.q.next.event.pointer
		BOOL x.q.full

where *x* is either *input*, *output* or *state* corresponding to queue data structures for the input, output and state queues respectively. *x.q.size* is a constant set up at compile time by definition in library *twlib.tsr* (logical name *ps.twlib*). *x.queue.n.states* are constants defined in the same library, which in the current race track implementation are set equal to 6.

The first 6 integer arrays define the input, output and state queue data contents. **.q.rec.time* and **.q.send.time* store event receive and send times respectively, and in the IQ both fields are set equal to the event receive time, as at this point the send time from the object is unknown. The next two arrays provide storage for the PID's of the sender and receiver of the event (*x.q.sender* and *x.q.recever*). For the IQ the receiver field should always be the PID of that object, and in the OQ the sender field will be equal to the local PID. The sign of the message, which dictates whether the message is an anti-message or not, is stored in *x.q.sign*, taking values of ± 1 only. Finally any information relating to the message object is stored in x.q.state, and where for example in the race track code, the 6 states correspond to vehicle parameters. Specifically vehicle ID, current vehicle PID location, the PID of the next object(not used), the current x and y coordinates and the vehicle velocity respectively.

This structure is shown in Figure 5 for an OQ. Chronological order of events is actually maintained with the array *output.q.pointer*, as described later.

The subsequent data structures maintain chronological order within the queues, and store other housekeeping information. For example, the i^{th} value of **.q.pointer* stores the location of the i^{th} ordered event. Throughout the simulation, events are constantly inserted and deleted from each of the queues, a link list is therefore provided containing pointers to the queue data fields, with entries corresponding to vacant positions in the structure. (*x.q.next.free*). As events are deleted from the queues, this list is updated ensuring the released storage space becomes available for the next event insertion. *next.free*, the head of the link list of free data structure positions, and *x.q.last* the reference to *x.q.pointer* containing the location of the final element in the queue are also updated.

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	2	5	9	10	13	14	16	22	-	-	-	-
Send Time	3	7	10	16	14	22	18	23	-	-	-	-
Sender	4	4	4	4	4	4	4	4	-	-	-	-
Receiver	2	5	6	3	3	5	5	6	-	-	-	-
Sign	+1	+1	+1	+1	+1	+1	+1	+1	-	-	-	-
State[0]												
State[1]												
State[2]												
..												

Figure 5: Queue Data Structure

For example, the send time of the i^{th} chronological event in the OQ is given by :-

output.q.send.time[output.q.pointer[i]]

the sign of the latest event in the IQ is at :-

input.q.sign[input.q.pointer[input.q.last]]

and the position of the next available vacant slot in the OQ data structure is at :-

output.q.pointer[output.q.next.free]

These structures are shown dynamically in the following example where for convenience only the receive time of the queue data structure will be shown. (- indicates undefined values)

The queue structures are initially set as in Figure 6.

An event arrives in the empty queue with a receive time of 100, which is inserted at the position given by *next.free* - Figure 7.

Subsequently event 2 arrives with a receive time of 85 (Figure 8).

Events 3 and 4 arrive to give the structure depicted in Figure 9.

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Free List	1	2	3	4	5	6	7	8	9	10	11	12
Queue Pointer	0	0	0	0	0	0	0	0	0	0	0	0

Next Free	0
Last Event	-1
Event Pointer	

(see later)

Figure 6: Initial Queue Structure Values

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	100	-	-	-	-	-	-	-	-	-	-	-
Free List	-	2	3	4	5	6	7	8	9	10	11	12
Queue Pointer	0	-	-	-	-	-	-	-	-	-	-	-

Next Free	1
Last Event	0
Event Pointer	-

(N.B. Only relavent values are given)

Figure 7:

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	100	85	-	-	-	-	-	-	-	-	-	-
Free List	-	-	3	4	5	6	7	8	9	10	11	12
Queue Pointer	1	0										

Next Free	2
Last Event	1
Event Pointer	-

i.e. last event at queue pointer[last event]

Figure 8:

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	100	85	106	82	-	-	-	-	-	-	-	-
Free List	-	-	-	-	5	6	7	8	9	10	11	12
Queue Pointer	3	1	0	2	-	-	-	-	-	-	-	-
Next Free	4											
Last Event	3											
Event Pointer	-											

Figure 9:

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	100	85	-	82	-	-	-	-	-	-	-	-
Free List	-	-	4	-	5	6	7	8	9	10	11	12
Queue Pointer	3	1	0	-	-	-	-	-	-	-	-	-
Next Free	2											
Last Event	2											
Event Pointer	-											

Figure 10:

The antimessage with receive time of 106 now arrives - this annihilates its positive message resulting in Figure 10.

The next two subsequent events will be inserted at positions 2 and 4 respectively.

Using the technique, storage space released from deleted events is always given priority for future use, therefore maintaining efficient use of the queue data structure memory allocation.

6 Message Passing Techniques

Here the techniques to implement message routing and message broadcasting will be described, together with a description of three low level routines used for message passing.

6.1 Message routing

6.1.1 Routing Table setup

Given an arbitrary network of M processes, each with up to L I/O channels (in this implementation L = 4), it is required to set up a vector for each process such that the i^{th} element contains the address (0 - L) of the output channel on which data should be sent to achieve a shortest path communication to process i. Therefore each intervening process adopts a similar strategy, of analysing the message destination field and routing as appropriate.

The definition of this route vector is easiest seen with reference to a simple example. Rather than describing individual vectors for each process, for convenience each vector will form part of a total routing table, such that the j^{th} row represents the routing vector in the j^{th} process.

In the current implementation the user is required to set up a table defining inter process connectivity. This is referred to as *conn.table* and is defined in the library *connecti.tsr* with logical name *ps.connections*.

For example, consider the following process network given in Figure 11.

This network has the following connectivity table (*conn.table*)

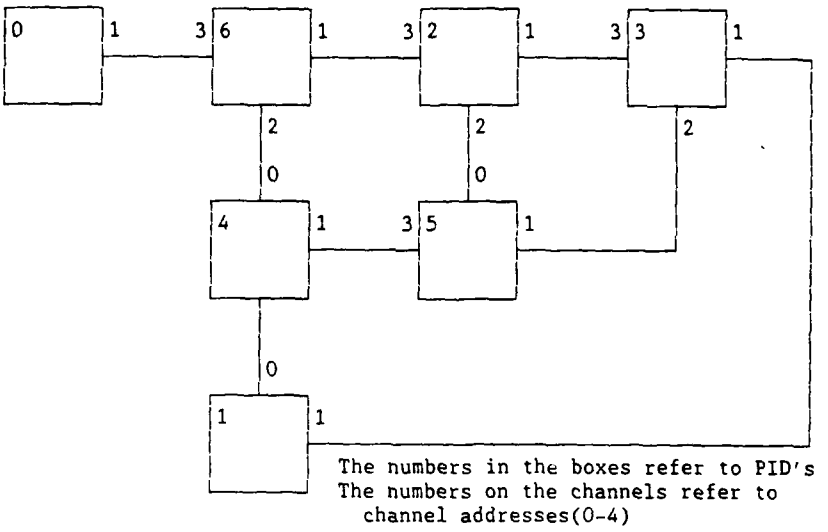


Figure 11: Example Processor Network

0	F	T	F	F
1	T	T	F	F
2	F	T	T	T
3	F	T	T	T
4	T	T	T	F
5	T	T	F	T
6	F	T	T	T

Where the first column indicates the connectivity of the North channel and so on.

Initially each process will have an undefined route vector. These become defined iteratively (maximum of *n.procs* iterations), by repeatedly exchanging, in parallel, data between all neighbouring processes along every connected channel (given by rows in the connectivity table), such that at the i^{th} iteration, each process has knowledge of all processes at a distance of i steps from itself. An example of this given below. Each row of the table represents the routing vector of the corresponding process.

Initial Conditions

	TO						
	0	1	2	3	4	5	6
0	-	*	*	*	*	*	*
1	*	-	*	*	*	*	*
2	*	*	-	*	*	*	*
3	*	*	*	-	*	*	*
4	*	*	*	*	-	*	*
5	*	*	*	*	*	-	*
6	*	*	*	*	*	*	-

Iteration 1

	TO						
	0	1	2	3	4	5	6
0	-	*	*	*	*	*	1
1	*	-	*	1	0	*	*
2	*	*	-	1	*	2	3
3	*	1	3	-	*	2	*
4	*	2	*	*	-	1	0
5	*	*	0	1	3	-	*
6	3	*	1	*	2	*	-

Iteration 2

	TO						
	0	1	2	3	4	5	6
0	-	*	1	*	1	*	1
1	*	-	1	1	0	0	0
2	3	1	-	1	2	2	3
3	*	1	3	-	1	2	3
4	0	2	0	2	-	1	0
5	*	1	0	1	3	-	0
6	3	2	1	1	2	1	-

Iteration 3

	TO						
	0	1	2	3	4	5	6
0	-	1	1	1	1	1	1
1	0	-	1	1	0	0	0
2	3	1	-	1	2	2	3
3	3	1	3	-	1	2	3
4	0	2	0	2	-	1	0
5	0	1	0	1	3	-	0
6	3	2	1	1	2	1	-

At this point the route table is complete and the setup terminates (As the maximum number of iterations is the maximum data path length, i.e. the network diameter). The table is actually distributed over the process network, with each process having only its own route vector (row from the above table), stored in the array variable *route.tab*. It should be noted that no attempt has been made to ensure an even distribution of

messages throughout the network, certain channels may carry a high traffic load. Also within the table setup procedure, channels are examined in numerical order, resulting in certain preferential routes. For example in the above network, data from process 5 to process 0 will always go via process 2 and not process 4, and similarly for data travelling in the reverse direction. If a more even balance of communications is desired a more intelligent routing table setup strategy will be needed.

A message routing sub process exists within each process, which either routes messages on to adjacent processes, or routes the message to its own internal process. This requires that each message carry a destination data field containing the process ID of the intended message recipient. As messages are received the following action is taken:-

```
If
    message.destination.field = PID
            then
                route message to own internal process
            else
                route message out on channel given by
                route.tab[message.destination.field]
```

This subprocess runs in parallel with the corresponding main internal process.

6.1.2 Message Broadcasting

Message broadcasting is essential to compute GVT at periodic intervals. A strategy has been developed for message broadcasting, which although not efficient, is robust and easy to implement. It is inefficient in that more than double the number of messages are transmitted in the system than are theoretically required.

A data broadcast consists of a broadcast initiator (usually the CP) and broadcast data recipients. In this simple implementation the broadcast strategy is common to both initiator and recipient.

The broadcast requires M steps, where M is the diameter of the data network. At each step, all processes await on receipt of the broadcast message, and if and when received, re-transmits it, on all connected links, including the link along which it was received. The processes then wait until one, and only one message has been transmitted and received (although not necessarily in that order) along each connected link. When this is complete for all connected channels for a process the message is transmitted to the internal process and appropriate action taken.

An example of this data broadcast technique is given in Figure 12 where we assume process 1 is the broadcast initiator. The numbers adjacent to each channel and process refer to the iteration in which data is transmitted along each channel. Thus, for example,

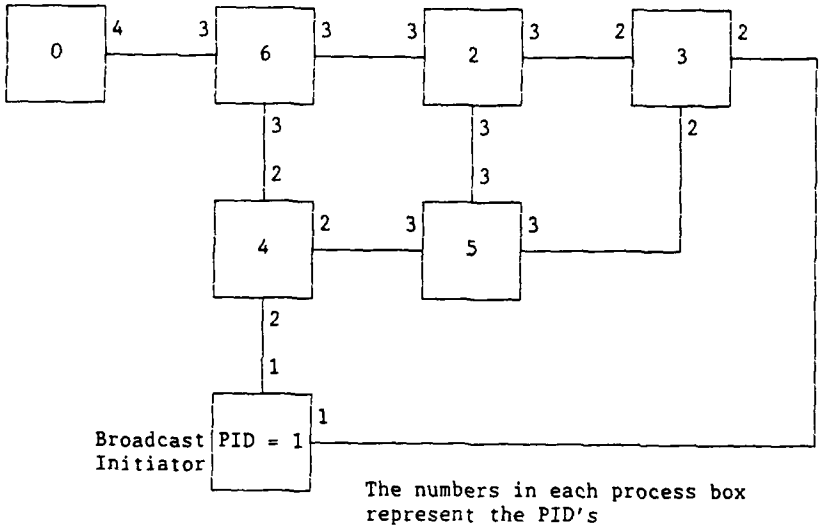


Figure 12: Data Broadcast Mechanism

in iteration 1, data is transmitted along channel 0 (to the north) and channel 1 (east) from process 1 and received in processes 3 and 4. At iteration 2 process 4 re-transmits this data along channels 0, 1 and 2 (North, East and South) to processes 6, 5 and 1 respectively. The final iteration is the 'echo' of data received in process 0 back to process 6.

6.2 Low level message passing procedures

Three simple low level procedures to allow message buffering and different forms of channel access are used extensively throughout the simulation code. These are store in library *net.tsr*, logical name *ps.net.procs*.

6.2.1 Procedure route

This routine is used from within the following two procedures and in the SCP. The routine provides a channel buffer defined with the ps.time.warp protocol, and handles all possible protocol combinations.

Parameters

1. **CHAN OF ps.time.warp input**

Input channel to buffer procedure.

2. **CHAN OF ps.time.warp output**
   Output channel from buffer process.
3. **CHAN OF INT stop**
   On receipt of any integer down this channel, the procedure (process) terminates.

6.2.2 Procedure array.in.four.out

This procedure accepts data from an array of 4 input channels, and outputs to one of four separately defined output channels. The procedure is used to convert output from each time warp process (an array of channels) to 4 distinct channels, to allow easier link and channel configuration. i.e. data on input[1] is output on channel output1 and so on. Each of the 4 internal data buffers run in parallel, so data blocking cannot occur.

Parameters

1. **[4]CHAN OF ps.time.warp input**
   The array of 4 input channels
2. **CHAN OF ps.time.warp output0**
3. **CHAN OF ps.time.warp output1**
4. **CHAN OF ps.time.warp output2**
5. **CHAN OF ps.time.warp output3**
   The four distinct output channels
6. **CHAN OF INT stopper**
   On receipt of an integer along this channel, the procedure terminates.

6.2.3 Procedure four.in.array.out

This procedure is identical to the previous except that it has four separate input channels and an array of four output channels.

Parameters

1. **CHAN OF ps.time.warp input0**
2. **CHAN OF ps.time.warp input1**

3. **CHAN OF ps.time.warp input2**

4. **CHAN OF ps.time.warp input3**

5. **[4]CHAN OF ps.time.warp output**

6. **CHAN OF INT stopper**

   As above.

6.2.4 Procedure analyse.network

This process is called from within each SP and the CP in the system, before entering the simulation phase, and returns important information relating to message routing, and process connectivity. It must not be called from the HP, GP or the SCP.

Parameters

1. **VAL INT proc.id**

   This is a unique process ID of the calling process and must be greater than or equal to zero. (Zero corresponds to the CP PID). It is used to address the user defined connectivity table (*conn.table* - which is defined within library *connecti.tsr* - logical name *ps.connections*. Unchanged on exit.

2. **VAL INT n.proc**

   This is the total number of processes in the system, excluding the HP, the GP and SCP. i.e it is the total number of SP's plus one (the CP).

3. **[4]CHAN OF ps.time.warp in**

   These four channels are the inputs to the respective process. In the current implementation, a maximum of 4 input and output channels have been permitted. Although 4 channels are declared, not necessarily all are used.

4. **[4]CHAN OF ps.time.warp out**

   As above, but the corresponding output channels.

5. **[]INT route**

   On return, the i^{th} value of this vector, contains the channel reference (0-3), onto which data should be transmitted to eventually arrive at the ith process. The data packet must therefore contain the destination PID address so that intervening processes can perform similar appropriate routing functions.

6. **[4]BOOL active.links**

   On return, values in this boolean vector indicate the connectivity of the 4 communication channels. i.e a value of TRUE in the i^{th} position means that communication

channel i is physically connected to another process. The vector is simply a copy of the relevant row of the interprocess connectivity table given by array *conn.table* as defined in the library *connecti.tsr* - logical name *ps.connections*.

6.3 Queue Handling Procedures

Five low level queue handling procedures have been produced providing facilities for queue initialisation, event insertion and deletion, and queue event selection. In the current implementation, the routines have not been written with efficiency in mind, if performance does become a crucial issue in the future, then large gains could possibly be made by rewriting these procedures.

These procedures reside in the library *qprocs.tsr* logical name *ps.q.procs*.

6.3.1 Procedure initialise.q

This procedure initialises the relevant queue pointers and is called once for each queue data structure, before the simulation starts It has 7 parameters.

1. **[]INT queue.time**

   This is the receive time field of the queue data structure, and is returned with all elements set to -1.

2. **[]INT queue.pointer**

   On exit all elements of this array are set to zero.

3. **[]INT queue.free.list**

   On exit, the i^{th} element of this array is set equal to $i + 1$. The final element is set to a constant identifying the end of list (*last.in.free.list*).

4. **INT last**

   Returned with a value of -1.

5. **INT next.free**

   Returned with a value of 0.

6. **BOOL queue.full**

   Returned set to FALSE.

7. **INT event.pointer**

   Returned with a value of -1.

6.3.2 Procedure find.pos.to.insert.in.queue

This procedure establishs the position in the queue data structure at which a new event message is to be stored. New events are only inserted if the queue is not already full. The routine primarily updates relevant queue pointers.

Parameters

1. **VAL []INT queue.time**

   This array contains the queue receive time. It is returned unchanged and is used to ensure queue data structure events are maintained in chronological order.

2. **[]INT queue.pointer**

   This array is used to store pointers to the chronological sequence of events in the queue data structure. It is returned with values updated ensuring that the new inserted event falls correctly in the chronological sequence of parameter 1.

3. **VAL []INT queue.free.list**

   The appropriate value is taken from *queue.free.list* to define the pointer to the next available free position in the queue data structure. The array is unchanged on exit.

4. **INT last**

   This is updated (incremented) to provide a reference to queue.pointer which points to the final event in the queue data structure.

5. **INT next.free**

   On entry this parameter contains a pointer to the next available vacant slot in the queue data structure. On return, it takes the next value from the link list given in free.list, i.e. on return it takes the value :-

   $$\text{next.free(return)} := \text{free.list[next.free(entry)]}$$

6. **INT event.pointer**

   Throughout the simulation, events are selected from the IQ and OQ. The position of this selection is given by event.pointer (A separate pointer for each queue). This value is a reference to queue.pointer, and is updated (incremented by one) if the new event insertion occurs earlier in the queue than the current selected event.

7. **INT insert.pos**

   On return this integer contains a reference to queue.pointer containing the insertion position in the queue data structure of the new event to maintain chronological order.

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	10	8	6	4	12	2	-	-	-	-	-	-
Free List	-	-	-	-	-	-	7	8	9	10	11	12
Queue Pointer	5	3	2	1	0	4						
Next Free	6											
Last Event	5											
Event Pointer	1											

```
Current event time
 = Receive Time[Queue pointer[Event pointer]]
```

Figure 13: Event Message Insertion

8. **INT new.time**

   On entry *new.time* contains the receive time of the event to be inserted in the queue data structure. The TW process requires events storage in chronological order. In this implementation the procedure searches along the queue from the earliest to the latest event to establish the correct insertion position, which may be inefficient if the queues are particularly long. It may therefore be desirable to rewrite this procedure if overall efficiency becomes important.

9. **BOOL queue.full**

   This is set TRUE if the queue data structure is full after event insertion.

For Example

We wish to insert an event with a receive time of 3 in the following queue data structure, where the current event selected has a receive time of 4 (i.e. event pointer = 1). Before insertion the queue data structure is as given in Figure 13 and after insertion as shown in Figure 14.

6.3.3 Procedure insert.in.q

Having previously made a call to *find.pos.to.insert.in.q*, to locate the reference to the insertion position via parameter 7(*insert.pos*), this procedure physically write the event data into the queue data structure. It is called once for each data field in the queue data structure. The procedure contains only one line of code, i.e.

queue[position] := value

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	10	8	6	4	12	2	5	-	-	-	-	-
Free List	-	-	-	-	-	-	-	8	9	10	11	12
Queue Pointer	5	6	3	2	1	0	4	-	-	-	-	-
Next Free	7											
Last Event	6											
Event Pointer	2											

Figure 14: Event Message Insertion

Parameters

1. **[]INT queue.field**

   This is the relevant field of the queue data structure in which 'value' is written.

2. **INT position**

   This is the reference to the queue data structure at which point 'value' is to be written. It is unchanged on exit. N.B. This is not the value returned in *insert.pos* after the call to *find.pos.to.insert.in.q*, but is the value referenced by it in *queue.pointer*.

   position := queue.pointer[insert.pos]

3. **INT value**

   This is the actual data value written in the relevant field of the queue data structure.

6.3.4 Procedure try.to.annihilate

This procedure is used to delete events from the queue data structures. In Time Warp simulation, events are only deleted when an antimessage arrives in a queue already containing its positive message, and during fossil collection. The procedure takes as parameters the queue data structure and the antimessage, and sequentially searches through the queue from the earliest to the latest event, until it finds the corresponding positive message, for large queues this may be inefficient and it may be appropriate to rewrite this procedure if efficiency ever becomes a prime concern. In this Time Warp implementation, the positive message will always arrive in the queue before its

corresponding antimessage, therefore events should always be annihilated. It is therefore a sensible precaution to check that deletion i.e. annihilation has occurred after a call to this routine.

During fossil collection, events earlier than GVT are deleted from the data queues. This procedure is also used to achieve this by artificially forming the antimessage of the event it is required to delete and calling this procedure. Again more efficient code could be written to implement this.

Parameters.

1. **[]INT queue.time**

   This array contains the receive time of the appropriate queue data structure, in which it is required to annihilate an event. It is unchanged on exit, and as the appropriate queue pointers are updated, there is no need to physically delete entries from this field.

2. **VAL []INT queue.from**

   As above, but contains the data field corresponding to the sender of the event.

3. **VAL []INT queue.sign**

   As above, but contains the data field corresponding to the sign of the message (either ± 1)

4. **[]INT queue.pointer**

   On exit this array is updated to ensure that chronological order of events is maintained.

5. **[]INT queue.free.list**

   On exit the relevant element of this array is set to the old 'next.free'. This effectively adds the vacant position to the front of the link list referring the remaining vacant positions in the queue structure.

6. **INT last**

   On exit this parameter is decremented, provided an annihilation occurs. This parameter references queue.pointer which returns the location of the last queue value.

7. **INT next.free**

   On exit this is set to the position of the annihilation, and is the position where the next insertion will occur.

8. **INT event.pointer**

   On exit, if an event is annihilated earlier than the current selected event given by event.pointer, this is decremented by one.

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	10	8	6	4	12	2	-	-	-	-	-	-
Free List	-	-	-	-	-	-	7	8	9	10	11	12
Queue Pointer	5	3	2	1	0	4	-	-	-	-	-	-
Next Free	6											
Last Event	5											
Event Pointer	3	(i.e. current event has receive time = 8)										

Figure 15: Event Message Annihilation

9. **VAL INT new.time**

   This integer is the receive time component of the message to be annihilated. Unchanged on exit.

10. **VAL INT new.from**

    As above but this parameter is the sender field of the message to be removed. Unchanged on exit.

11. **VAL INT new.sign**

    Events will only be deleted if new.time and new.from exist within the queue data structure and if new.sign has the opposite sign to that located in the queue data structure. Unchanged on exit.

12. **BOOL annihilated**

    This flag is set TRUE if a valid annihilation occurs, i.e. if new.time and new.from are located within the queue data structure and new.sign has the opposite sign to that located within the data structure.

13. **INT ani.pointer**

This returns the reference to queue.pointer, containing the location of the annihilated event. It is used to reset a pointer during rollback at the OQ stage. For example we wish to remove an event with a receive time of 4 as indicated in Figure 15, which results in Figure 16.

	0	1	2	3	4	5	6	7	8	9	10	11
Receive Time	10	8	6	-	12	2	-	-	-	-	-	-
Free List	-	-	-	6	-	-	7	8	9	10	11	12
Queue Pointer	5	2	1	0	4	-	-	-	-	-	-	-

Next Free	3
Last Event	4
Event Pointer	2

(i.e still refers to event with Receive Time = 8)

Figure 16: Event Message Annihilation

6.3.5 Procedure select.next.q.event

This function takes as part of its parameter list the current selected event, and returns a pointer to the next chronological event. It is used to select events from the IQ for processing and to select events from the OQ for output.

Parameters

1. **VAL []INT queue.time**

   This is the receive time field of the appropriate queue data structure. It is unchanged on exit

2. **VAL []INT queue.pointer**

   This is the pointer array providing chronological access to the queue data structures. Unchanged on exit.

3. **VAL []INT queue.free.list**

   The array containing the link list of vacant positions in the queue data structure. (N.B. This array is not used by this procedure and is returned unaltered).

4. **VAL INT last**

   This integer acts as a reference to queue.pointer to the last event in the queue data structure. Unchanged on exit.

5. **VAL INT next.free**

   An integer pointing to the next free available position in the queue data structure. (N.B. Not used in the current version of this routine)

6. **VAL INT this.lvt**

   On entry this parameter contains the current value of the event receive time. The next event in the chronological sequence is returned. To select the first event in the queue, this parameter and *event.pointer* should both be set to -1. (e.g. to indicate the start of fossil collection).

7. **INT event.pointer**

   On entry this parameter contains a pointer to the currently selected event. The search for the next event starts from this event, making the procedure relatively efficient. Without this parameter the search for the next event would have to start from the beginning (or end) of the chronological sequence and work towards the end (beginning). Also it is perfectly valid for events to exist in the event queues with identical receive times, which without this pointer could result in events being selected more than once. On exit this pointer is updated to point to the next event in the queue data structure.

8. **BOOL selected**

   This flag is set to TRUE if the procedure successfully selects the next event from the queue data structure, else, if the selection goes off the end of the queue, is set FALSE.

6.4 Message Protocols

Within the Time Warp simulation system, extensive use is made of occam 2 channel protocols. All interprocess messages conform to the complex protocol defined by *ps.time.warp*. The definition of which is stored in the library *protocol.tsr* (logical name *ps.protocols*).

6.4.1 Protocol ps.time.warp

All transputer links and interprocess channels (within a processor) are given this general protocol, which currently allows 30 message types. (A few are redundant or obsolete).

Note Changes to the protocol definition should be made sparingly, since it necessitates recompilation of the entire time warp system.

In the following section, a description of each protocol is provided. Each protocol data field will be referred to as P1, P2, etc corresponding to the first, second etc message component of the respective protocol.

1. **event.road.p ; INT ; INT ; INT ; INT ; INT**

   No longer used

2. **event.route.p ; INT ; INT ; INT ; INT ; INT ; INT :: []INT**

   Every inter process simulation event message in the system obeys this protocol, including positive and anti-messages. P1, P2, P3 , P4 and P5 correspond to the sender PID, receiver PID, message send time, message receive time and message sign (± 1) respectively. The final vector structure, given by P6 and P7 allow transmission of any special message components. For example in the 'race track' example, P7 has 6 components corresponding to vehicle ID, vehicle source, vehicle destination, vehicle position (x and y coordinates) and velocity.

   This protocol is not used however between the time warp process and the message router.

3. **gvt.broad.p ; INT ; INT**

   This is used to broadcast GVT from the CP to each SP during the GVT estimation phase. P1 signifies the ID of the process initiating the broadcast - in this case the CP (ID = 0), and P2 is the computed GVT.

4. **gvt.req.broad.p**

   Again a broadcast protocol which is part of the GVT estimation procedure. The gvt.req.broad.p signal is generated by the controller at regular physical times (given by *gvt.freq*) and sent to all SP's, to request each to stop outputting and flush all output buffers (i.e. await acknowledgements of all previously sent messages).

5. **ok.to.cont.broad.p**

   Used by the controller to broadcast a signal to all SP's, allowing them to continue normal processing following GVT estimation and fossil collection.

6. **gvt.pause.ack.p ; INT ; INT**

   This signal is sent by each SP to the CP, to signify that the SP has ceased outputting events, and that all previously sent messages have been received. P1 and P2 represent 'from' and 'to' fields of which 'to' is required by the data routing processes. Here P1 is the sender PID and P2 is 0 (the CP PID).

7. **lvt.ack.p ; INT ; INT ; INT**

   Again part of the GVT estimation protocol, here used to send the LVT of each SP back to the CP. P1 and P2 are the 'from' and 'to' message routing fields, and P3 the LVT.

8. **lvt.req.broad.p**

   After all the SP's have suspended output following a *gvt.req.broad.p* request, the CP broadcasts this signal to all SP's to request their respective LVT's.

9. **event.p ; INT ; INT ; INT ; INT ; INT ; INT :: []INT**

   This is the message protocol used between the sequential TW process and the I/O data router. Message components are as defined in the *event.route.p* protocol above.

10. **fc.event.route.p ; INT ; INT ; INT ; INT ; INT :: []INT**

    During Fossil Collection, events are deleted from each OQ in all the SP's up to (but not including GVT). In addition these events are routed to the CP with this protocol for snapshot generation. Here the message components are as defined in *event.route.p*. Since to TW methodology guarantees that rollback will not occur prior to GVT, the sign field should always be +1.

11. **fc.end.of.event.p ; INT ; INT**

    Used by each SP to denote to the CP that all Fossil Collection events have been transmitted. P1 and P2 are the 'from' and 'to' fields required for data routing.

12. **i1.p ; INT**

    Not Used.

13. **i2.p ; INT ; INT**

    Used by the HP to send the output option (P1) and graphics delay (P2) to the CP.

14. **i3.p ; INT ; INT ; INT**

    Not used.

15. **i4.p ; INT ; INT ; INT ; INT**

    Not used.

16. **i5.p ; INT ; INT ; INT ; INT ; INT**

    Not used.

17. **iv.p ; INT :: []INT**

    This protocol is currently used to pass initialising events from the HP to the CP. P1 is the size of P2, which in the current implementation contains initial vehicle parameters. i.e. start time, vehicle type, source PID of vehicle, destination (set equal to source), and velocity.

18. **an.iv.p ; INT :: []INT**

    This is used during the analyse network phase only (*analise.network*) to exchange neighbourhood data between connected processes, to set up the message routing tables.

19. **an.i1.p ; INT**

    Not used.

20. **event.rec.ack.p ; INT ; INT**

    As each event message is received by the intended recipient an acknowledgement of receipt is generated and returned to the message sender. P1 and P2 are used for message routing to indicate 'from' and 'to' fields with respect to the direction of the message acknowledgement.

21. **mess.ack ; INT**

    Not used.

22. **proc.term.p**

    Not used.

23. **ps.prot.tt.out.string ; INT ::[]BYTE**

    This is used to pass data from the CP to the HP and host screen. Screen messages are encoded using derivatives of the Inmos I/O routines (specifically ps.newline, ps.write.int, ps.write.len.string and ps.write.full.string). These are necessary, since the current occam io library routines do not permit the use of channels with protocols.

24. **ps.stop**

    Not used.

25. **start.and.analise**

    This signal protocol is sent from the HP to the CP denoting permission for the SPA to enter the initialisation and analysis phase to set up routing tables.

26. **start.and.analise.ok**

    Not used.

27. **start.of.event.list**

    A protocol signal sent by the HP to the CP to indicate that subsequent messages along the channel are initialisation events.

28. **end.of.event.list**

    As above, but signifies the end of the initialisation events.

29. **gr.raster ; [512]BYTE**

    Used to transmit graphics data from the HP to the GP running in the B007 processor. This data is transmitted via the CP. In the race track example program, the graphics backdrop is read to the GP with this protocol.

30. **gr.setup.ok**

    A simple signal protocol used to indicate to the CP that the graphics backdrop has been successfully sent and displayed on the GP.

6.5 Libraries

The current implementation uses 8 libraries containing commonly used procedures and various compile time parameters. Library access is via the #USE occam construction with either the library physical file name or logical name as a parameter. Below is given a description of each library with its contents.

1. **protocols.tsr - logical name ps.protocols**

   Contains the definition of the protocol *ps.time.warp* only.

2. **twlib.tsr - logical name ps.twlib**

   Contains constant definitions relating to the queue data structure sizes together with other constants pertaining to the current time warp implementation. The only constants which may require change are *event.size*, the number of states associated with event messages - currently 6, *host.event.size*, the number of parameter used to define initialisation events - currently 5 and *max.no.not.acked* the maximum number of un-acknowledged event messages permitted in transit from one process to another. There is nothing to be gained by increasing this parameter, and under no circumstances should it be increased beyond the number of inter TW buffers which is currently two. Reducing it to zero may result in an overall decrease in performance..

3. **gvt.tsr - logical name ps.gvt**

   Contains the definition of *gvt.freq* only. i.e. the rate at which GVT estimation and fossil collection should be undertaken. *gvt.freq* should be expressed in milliseconds.

4. **roadlib.tsr - logical name ps.road.lib**

   This library contains the physical coordinates of the road network. In the race track demonstration, the road is split into 38 straight line segments. The coordinates defined in *track* represent road segment end coordinates.

5. **q.procs.tsr - logical name ps.q.procs**

   This library contains the 5 occam procedures used for manipulating the time warp queue data structures. Specifically *find.pos.to.insert.in.q* identifies the insertion position of new events and adjusts pointers to maintain chronological order within the structures. Procedure *insert.in.q* actually copies new events into the queue data structure at the position returned by the previous procedure. Procedure *try.to.annihilate* attempts to cancel an event message with its 'anti' counterpart i.e the same message but with opposite sign. *Initialise.q* initialises the queue data structures and associated pointer arrays, and procedure *select.next.q.event* selects the next chronological event (either for processing from the OQ or output from the OQ) given the queue position and time of the previous event message.

6. **connecti.tsr - logical name ps.connections**

   This library contains the definition for the connection table *conn.table* providing user defined data relating to the current configuration being used. It is extremely important that this structure is set correctly as incorrect values will result in routing tables incorrectly defined causing unpredicted event message routing, probably giving rise to deadlock. The table is arranged as a two dimensional structure with each row containing 4 boolean entries, corresponding to the connectivity of each of the 4 process channels. The i^{th} row is the connectivity for the i^{th} TW process, with the only constraint that the CP is process 0. Channels 0 and 3 of the CP are connected to the HP and GP respectively and take no part in event message routing. Therefore these corresponding entries MUST be set FALSE, even though they are connected to other processes.

7. **psuserio.tsr - logical name ps.userio**

   This library contains 4 procedures (*ps.write.len.string*, *ps.write.full.string*, *ps.newline* and *ps.write.int*) that are identical to their Inmos counterparts (i.e. without the *ps.* suffix) with the exception that output conforms to the *ps.prot.tt.out.string* protocol, which is part of the *ps.time.warp* complex protocol.

8. **net.tsr - logical name ps.net.procs**

   This library contains various procedures associated with data routing. These are *route*, *array.in.four.out* and *four.in.array.out*, used to provide either access to distinct channels or arrays of channels and *analise.network*, used to set up routing tables given process connectivity data as defined in structure *conn.table*.

7 Error Conditions

Certain checks are made throughout the time warp simulation process to trap invalid conditions. Currently these conditions cause the transputer error flag to become set, via procedure *crash* resulting in abnormal termination of the processor array. The location of these errors, if they occur can readily be located using the TDS debugger. During normal simulator use these error conditions will not occur.

Trapped Fatal Error Conditions

In the CP

A fatal error can only occur in the CP if a message is received with an unexpected protocol.

In the SP's

Input router - an error will occur only if a message with a forbidden protocol arrives.

Time warp process - the following will cause the transputer error flag to become set.

- If an incorrectly routed message arrives.
- If the IQ becomes full.
- If an attempt is made to insert an antimessage into the IQ - normally these should annihilate with a previously stored positive message.
- If duplicate events occur in the IQ.
- If a selected rollback event is not annihilated from the OQ.
- If the OQ becomes full.
- If duplicate events occur in the OQ.
- If duplicate events are sent to the CP during fossil collection.
- If an event is not removed from the IQ during fossil collection.
- If an event is not removed from the OQ during fossil collection.

8 Implementation Constraints

The current Time Warp implementation has a number of restrictions. It would be desirable, although not essential to remove these from future implementations (with the exception of the first below).

- The current data routing algorithm is not robust, and deadlock will occur if events are routed randomly within the processor array. In the current implementation all event messages are routed to neighbouring processes, with returned acknowledgements also therefore between neighbouring processes. This appears to be deadlock free. During fossil collection all events are routed to a single destination (the CP) which also appears to be robust.
- Currently it is not possible to route a message back to the input of the originating process. This can easily be implemented by providing a soft channel from the output routing subprocess back to the input process.
- Currently logic has been inserted in the occam code that accepts messages, to suppress the generation of acknowledgements to initialising events. A trivial fix to this is to allow the CP to accept acknowledgements.
- The system will crash when either of the input, internal state or output queues fill. Remedies to this are documented by Jefferson, e.g. returning messages if they can not be inserted in an input queue. Alternatively, estimating GVT more frequently will force more frequent fossil collection, or if memory prevails increasing the event queue sizes.

- During fossil collection, events are selected and deleted from the OQ and routed to the CP. The corresponding event is then removed from the IQ. If however an input event triggers two or more output messages, then currently the code will attempt to locate these subsequent events in the IQ and fail, since they are not there - resulting in the system stopping. Slightly more intelligent logic during event removal from the IQ during fossil collection is required to prevent this occurring.
- The event history array in the CP, used to store events for display has a finite size. If this fills, the system will terminate. It may be possible to dispense with this large data structure completely, or alternatively computing GVT more frequently will mean that fewer events are received during fossil collection.
- Currently the user has to manually configure the desired processor topology, allocate processes to them and set up a connectivity table. It would be desirable to do this automatically since it is extremely error prone.
- In future implementations the dynamic creation and deletion of simulation objects would be highly desirable. It is unclear at the moment how this could be achieved.
- GVT estimation requires a complex message exchange protocol, to ensure that all processes shut down in a controlled fashion. It may be possible to use the Supernode's control bus to facilitate this.

9 Race Track Traffic Flow Example

A demonstration example based on traffic flow modelling has been written to prove the time warp concept and validate the TW code. The example is based on vehicles travelling at different velocities around a figure of eight race track, in which road segments are represented as stationary objects resident in each SP and vehicles as event messages moving from object to object. Associated with each event message therefore, are vehicle properties including vehicle position and which is updated as vehicles move from road segment to road segment and vehicle identification and velocity. It is assumed in this demonstration that vehicles do not interact with each other, so that vehicles may coexist at the same point in space and time, and are transparent to each other for overtaking purposes.

Road segment objects therefore take event messages representing vehicles as input, modify vehicle states (e.g. position and time), together with local road state changes (e.g. time) and output vehicles to the next process. Rollback is triggered each time a vehicle arrives in a road process with a time less than the local virtual time associated with that road segment process - which will be arrival time of its last vehicle. If vehicles with differing velocities are used for input, then considerable numbers of rollbacks will occur allowing rigerous system testing.

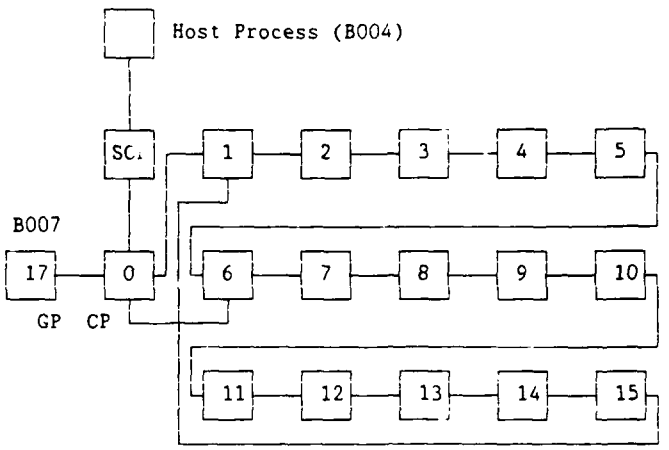


Figure 17: Race Track Processor Configuration

9.1 Input Data

Input data residing in a TDS fold consists of the graphics backdrop - a binary dump of a figure of eight race track, and a fold containing initial vehicles in the system. Each vehicle is represented by a line in this fold each having 5 integers, vehicle start time, vehicle ID, vehicle starting road segment, vehicle destination road segment (not used) and vehicle velocity respectively. The fold end is indicated by -1 as the last entry. Currently up to 1000 initial vehicles are permitted, although this limit can be increased by changing the value of the constant max.no.events, as defined in library *tw.lib.tsr* - logical name *ps.tw.lib*.

In the current implementation the coordinates of each of the 38 road segments defining the figure of eight road system are provided in the library *roadlib.tsr* - logical name *ps.road.lib*. In future implementations it may be more convenient to read this information in at run time, as currently, any changes to the network necessitates recompilation of the simulation system.

9.2 Configuration

The current processor configuration is as shown below in Figure 17, where physical transputer links 0, 1 , 2 and 3 are represented by lines from the upper, right, lower and left edges respectively of the boxes representing processors. Basically the network is configured as a circular ring with connections from the CP to processors 1 and 6. Odd and even numbered simulation transputers (1-15) contain 3 and 2 time warp simulation

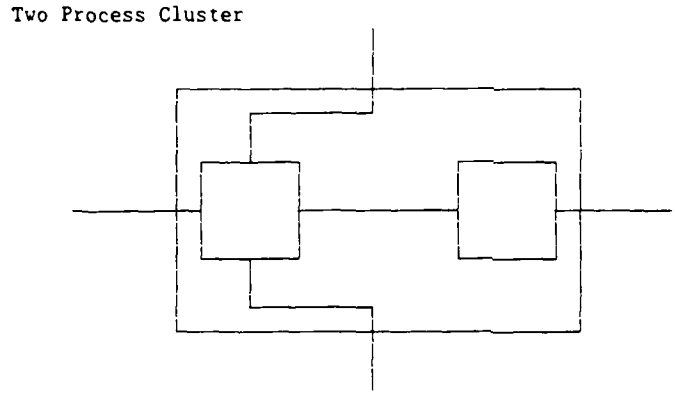


Figure 18: Two Process Cluster

processes respectively, such that adjacent road segments are represented in adjacent simulation processes. Given a non-blocking communications system however, this condition could be relaxed. The internal process configuration within processors for the 2 and 3 time warp simulation processes are shown in Figure 18 and Figure 19 respectively.

Entries in the process connectivity table conn.table refer to the connectivity with respect to simulation processes not processors and is as shown in Figure 20.

9.3 Memory Requirements

The amount of transputer memory required depends largely on the queue data structure array sizes. In this example, each simulation process has queue structures defined with capacities of up to 300 entries, and each requires about 60K Bytes of memory, permitting a maximum of three simulation objects per processor in the rat cage implementation where each transputer has 256K Bytes of memory. However the physical queue sizes required will very much depend on the application, it may be possible to reduce their size significantly. The CP requires 255K Bytes of memory with most of this taken up with an array storing the event history, required to provide an ordered sequence of events to the GP. If during normal use it is found that this array fills then it will be necessary to estimate GVT more frequently. The HP, GP and SCP use 30K, 477K and 4K Bytes respectively.

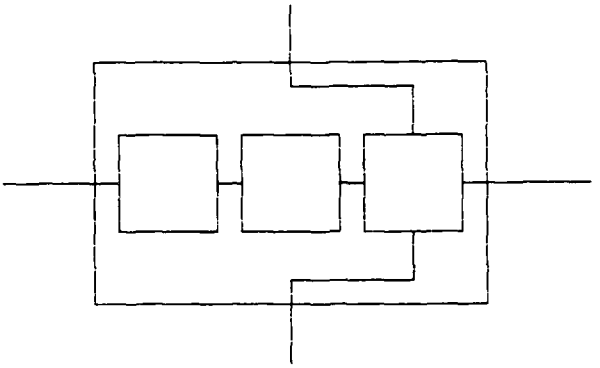


Figure 19: Three Process Cluster

10 The Way Ahead

This report has described the implementation of Jefferson's [1] Time Warp technique on a distributed array of transputers. Section 8 however describes a few limitations of the current system which should be addressed with varying priorities in future versions. The major limitation currently is in the routing strategy used, which is not robust enough to guarantee deadlock free operation for the random point to point communications required in a general time warp implementation. Future work must address this issue, which is also the topic of research in several laboratories. [2,3]. Additionally, the current version relies on a complex message protocol for estimating GVT. This could be simplified in future versions by exploiting global communications paths provided on many commercial transputer systems.

Longer term work must also address the need for a high level object orientated implementation language, and the interface between this and Time Warp and also investigate ways of measuring efficiency and obtain performance comparisons with other simulation techniques.

11 Conclusion

The successful application of transputers to distributed event driven simulation using the Time Warp methodology has clearly been demonstrated with transputers and occam providing a natural vehicle for this class of simulation. The simulation technique basically comprises a number of communicating simulation object processes, with appropriate action being taken to ensure the correct chronological sequence of processed simulation events. Time Warp is particularly attractive, since it permits all parts of

Process	channel			
	0	1	2	3
0	F	T	T	F
1	F	T	T	T
2	F	T	F	T
3	F	T	F	T
4	F	T	F	T
5	F	T	F	T
6	F	T	F	T
7	F	T	F	T
8	F	T	F	T
9	F	T	F	T
10	F	T	F	T
11	F	T	F	T
12	F	T	F	T
13	F	T	F	T
14	F	T	T	T
15	F	T	F	T
16	F	T	F	T
17	F	T	F	T
18	F	T	F	T
19	F	T	F	T
20	F	T	F	T
21	F	T	F	T
22	F	T	F	T
23	F	T	F	T
24	F	T	F	T
25	F	T	F	T
26	F	T	F	T
27	F	T	F	T
28	F	T	F	T
29	F	T	F	T
30	F	T	F	T
31	F	T	F	T
32	F	T	F	T
33	F	T	F	T
34	F	T	F	T
35	F	T	F	T
36	F	T	F	T
37	F	T	F	T
38	F	T	F	T

Figure 20: Race Track Processor Connectivity Table

a distributed network to operate in parallel (although some of the computation may later be undone). The need for hardware control of memory management has not been identified, although the need for a deadlock free, random point to point communications strategey has.

12 References

[1] Jefferson D, "Virtual Time", Int. Conf. on Parallel Processing, 1983, pp 384-394.

[2] Roscoe A.W., "Routing messages through networks: an exercise in deadlock avoidance", Proc. 7th occam Users Group and International Workshop on Parallel Programming of Transputer based Machines, Grenoble, Sept 14-16 1987, Ed. Traian Muntean.

[3] Muntean T., "PARX operating system kernal; application to Minix", Esprit P1085 "Supernode" Project, Work Package 5 Working Paper, Jan 1989.

13 Appendix

The following block diagrams provide a complete breakdown of the entire time warp system, including the HP, CP, GP and each CP.

Parallel and sequential processes are separated by vertical and horizontal lines respectively. Where one of several options can occur, for example following input to an occam ALT statement, options are separated with a dashed horizontal line.

The block diagrams are heirarcical, with composite processes resolved gradually into their respective atomic processes.

TIME WARP SIMULATION - complete overview

Host Process (HP)	Switch Controller Process (SCP)	Controller Process (CP)	Time Warp Simulation Processes (SP's)	Graphics Process (GP)
(Host B004)	(Proc 0)	(Proc 16)	(Procs 1-15)	(Proc 17)

TIME WARP SIMULATION - host process

Output Header Information				
File Output?				
Initialise Output Option				
Initialise Graphics Delay				
Route Output to Screen	Route Output to File	Multiplex Output to File and screen	Simulation initialisations	Receive messages from Simulation Array

TIME WARP SIMULATION - HOST PROCESS - simulation initialisations

Send start.and.analise signal to array
Read in graphics backdrop and send to array
Read in initial event list
Display event list
Send event list to array

TIME WARP SIMULATION - Switch Controller Process

Route data from Host (HP) to Controller (CP)	Route data from Controller (CP) to Host (HP)

TIME WARP SIMULATION - Controller Process

Convert 4 separate input channels to array of input channels	Convert array of output channels to 4 separate output channels	Controller Subprocess

TIME WARP SIMULATION - CONTROL PROCESS - controller subprocess

start.and.analyse	
	Send introduction message to host(HP)
	Analyse network
	Display CP route vector
Initialise Graghics process	
	Read backdrop from HP
	Send data to GP

Initialise initial event list		
	Read start.of.event.list from HP	
	Read event list from HP	
	Read end.of.event.list from HP	
Message Handler and Initialisation		
	Send event list to SPA	Main Simulation Controller

Main Simulation Controller

ALT	Generate GVT estimation Interupt
	Receive acknowledgements from CP initiated broadcasts
	Receive LVT's from SPA
	Receive acknowledgements that SP's have shut down for GVT estimation
	Receive Fossil Collection data
	Receive any diagnostics data - routed to HP

GRAPHICS PROCESS

Data Formatter		Graphics Controller	
	Input and format graphics Commands		Display Graphics data

TIME WARP SIMULATION PROCESS ARRAY

Time Warp object Process	Time Warp object Process	Time Warp object Process	Time Warp object Process	...

Time Warp Object Process

Convert 4 separate input channels to array of input channels	Convert array of output channels to 4 separate channels	Analyse Network
		Sequential Time Warp Process

Analyse Network

Initialise variables
Ascertain connected channels
Compute routing table

Sequential Time warp process

Initialisations	
Routing Process	Sequential time warp subprocess

Routing Process

Input Router	Output Router
Wait for input on any input channel and act according to message type. Pass data to output router or sequential time warp subprocess	Wait for data from input router or from sequential time warp subprocess, and output it on appropriate output channel

Sequential Time Warp Subprocess

Initialisations	
Compute Object Properties	
PRI ALT	Input to Time Warp Subprocess
	Output next event
	Process next event from IQ
	Acknowledgement for GVT shutdown
	Fossil Collection

Input to Time Warp Subprocess

event (event.p)	Send acknowledgement to sender
	Attempt to Annihilate from IQ
	If not annihilated the insert in IQ
	Test for Rollback
event acknowedgement (event.rec.ack.p)	Decrement counter of number of events in transit (i.e. events sent but not received)
request shutdown for GVT estimation (gvt.req.broad.p)	Set relavent flags
request for LVT	send LVT to CP
receive GVT from CP (gvt.broad.p)	Fossil Collection
fossil collection events received by CP (ok.to.cont. broad.p)	Resume normal Processing

Output next event

If rollback flag is TRUE	Select and delete appropriate messages from OQ, and form and send appropriate anti-messages
else	Select next positive message and send it to the output router

Process Event from IQ

Select next unprocessed event from IQ
Process this event and store any local state changes
Insert processed event in OQ

Acknowledgement that process has shut down for GVT estimation

Send message to CP signifying that all sent messages have been received, and that process has suspended outputting

Fossil Collection

Select all events from OQ earlier than GVT
Send these events to CP
Delete these events from OQ
Delete corresponding events from IQ
Send fc.end.of.event.p signal to CP to signify that all events have been sent

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification, eg (R), (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	MEMO 4334		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCESTERSHIRE WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

OBJECT ORIENTATED SIMULATION ON TRANSPUTER ARRAYS USING TIME WARP

7a. Title in Foreign Language (in the case of Translations)

7b. Presented at (for Conference Papers). Title, Place and Date of Conference

8. Author 1: Surname, Initials	9a. Author 2	9b. Authors 3, 4 . . .	10. Date	pp ref
SIMPSON P			1989.10	61

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution Statement

UNLIMITED

Descriptors (or Keywords)

Continue on separate piece of paper

Abstract

The successful application of Transputers to distributed event driven heterogeneous simulation using the Time Warp methodology is demonstrated with transputers and occam providing a natural vehicle for this class of simulation. The simulation technique basically comprises a number of communicating simulation object processes, with appropriate action being taken to ensure the correct chronological sequence of processed simulation events. Time Warp is particularly attractive, since it permits all parts of a distributed processor network to operate in parallel (although some of the computation may later be undone). The need for hardware control of memory management has not been identified, although the requirement for a deadlock free, random point to point communications strategy has.